RWTH AACHEN UNIVERSITY

AICES

Algorithms for Large-scale Whole Genome Association Analysis

Elmar Peise Diego Fabregat Yurii Aulchenko
Paolo Bientinesi

Aachen Institute
for Advanced Study in
Computational Engineering Science

Financial support from the
Deutsche Forschungsgemeinschaft (German Research Foundation)
through grant GSC 111 is gratefully acknowledged.

Algorithms for Large-scale Whole Genome Association Analysis

Elmar Peise
RWTH Aachen University
Aachen, Germany
peise@aices.rwth-aachen.de

Diego Fabregat
RWTH Aachen University
Aachen, Germany
fabregat@aices.rwth-aachen.de

Yurii Aulchenko
Institute of Cytology and Genetics
Novosibirsk, Russia

Paolo Bientinesi
RWTH Aachen University
Aachen, Germany
pauldj@aices.rwth-aachen.de

ABSTRACT

In order to associate complex traits with genetic polymorphisms, genome-wide association studies process huge datasets involving tens of thousands of individuals genotyped for millions of polymorphisms. When handling these datasets, which exceed the main memory of contemporary computers, one faces two distinct challenges: 1) Millions of polymorphisms come at the cost of hundreds of Gigabytes of genotype data, which can only be kept in secondary storage; 2) the relatedness of the test population is represented by a covariance matrix, which, for large populations, can only fit in the combined main memory of a distributed architecture. In this paper, we present solutions for both challenges: The genotype data is streamed from and to secondary storage using a double buffering technique, while the covariance matrix is kept across the main memory of a distributed memory system. We show that these methods sustain high-performance and allow the analysis of enormous datasets.

Keywords

genome-wide association study, mixed-models, generalized least squares, out-of-core, distributed memory, Elemental

1. INTRODUCTION

Whole Genome Association Studies, also known as Genome-Wide Association (GWA) studies, became the tool of choice for the identification of loci associated with complex traits. The association between a trait of interest and genetic polymorphisms (usually single nucleotide polymorphisms, SNPs) is studied using thousands of people typed for hundreds of thousands of polymorphisms. Thanks to these studies, hundreds of loci for dozens of complex human diseases and quantitative traits have been discovered [6]. In GWA analysis, one of the most used methods to account for the genetic substructure due to relatedness and population stratification is the variance component approach based on mixed models [2, 11]. While effective, mixed-models based methods are computationally demanding both in terms of data management and computation. The objective of this research is to make large-scale GWA analyses more affordable.

Computationally, a GWA analysis based on approximations to the mixed-model applied to a set of n individuals and m genetic markers (SNPs) boils down to the solution of m generalized least-squares (GLS) problems

$$b_i := \left(X_i^T M^{-1} X_i\right)^{-1} X_i^T M^{-1} y, \quad \text{with} \quad i = 1, \ldots, m, \quad (1)$$

where the $X_i \in \mathbb{R}^{n \times p}$ is the design matrix, $M \in \mathbb{R}^{n \times n}$ is the covariance matrix, the vector $y \in \mathbb{R}^n$ contains the phenotypes, and the vector $b_i \in \mathbb{R}^p$ expresses the relation between a variation in the SNP (X_i) and a variation in the trait (y). Additionally, M is symmetric positive definite (SPD), $2 \leq p \leq 20$, n ranges approximately between 10^3 and 10^5, and m ranges between 10^5 and 10^8. Finally, X_i is full rank and can be viewed as composed of two parts, $X_i = \left(X_L \middle| X_{Ri}\right)$, with $X_L \in \mathbb{R}^{n \times (p-1)}$ and $X_{Ri} \in \mathbb{R}^{n \times 1}$, where X_L is constant across all m genetic markers.

The first reported GWA study dates back to 2005: 146 individuals were genotyped, and about 103,000 SNPs were analyzed [7]. Since then, as the catalog of publishd GWA analyses shows [5, 9], the number of published studies has increased steadily, up to 2,404 in 2011 and 3,307 in 2012. A similar growth can be observed both in the population size and in the number of SNPs: Across all the GWAS published in 2012, on average, the studies used 15,471 individuals, with a maximum of 133,154, and 1,252,222 genetic markers, with a maximum of 7,422,970. From the perspective of Eq. (1), these trends present concrete challenges, especially in terms of memory requirements. As both $M \in \mathbb{R}^{n \times n}$ and the X_i's compete for the main memory, two scenarios arise: 1) If n is small enough, M fits in memory, and the X_i's have to be streamed from disk; 2) if M does not fit in memory, then both data and computation have to be distributed over multiple compute nodes. In this paper we present efficient algorithms for both scenarios.

Several notable implementations for GWA studies already exist: GenABEL is a widely spread library for genome studies [1]; FaST-LMM is a program specifically designed for

large datasets [8]; recently, a new high-performance implementation, to which we refer as SMP-OOC, was introduced in [3]. None of these algortihms are meant for distributed memory systems; hence, for all of them the population size n is limited by the memory of a single node.

The rest of this paper is structured as follows. A mathematical algorithm used to solve Eq. (1) is discussed in Section 2. Then, a technique to make the algorithm feasible for an arbitrary numbers of SNPs out-of-core is presented in Section 3. Finally, in Section 4, the algorithm is further extended to deal with large population sizes by means of distributed memory architectures.

2. THE MATHEMATICAL ALGORITHM

The standard route to solving one of the GLS's in Eq. (1) is to reduce it to an ordinary least squares problem (OLS),

$$b_i = \left(\overline{X}_i^T \overline{X}_i\right)^{-1} \overline{y},$$

through the operations

1	$LL^T \coloneqq M$	(Cholesky factorization)
2	$\overline{X}_i \coloneqq L^{-1} X_i$	(triangular solve)
3	$\overline{y} \coloneqq L^{-1} y$	(triangular solve)

The resulting OLS can then be solved by two alternative approaches, respectively based on the QR decomposition of $\overline{X}_i$, and the Cholesky decomposition of $\overline{X}_i^T \overline{X}_i$. In general, the QR-based method is numerically more stable; however, in this specific application, since $\overline{X}_i^T \overline{X}_i \in \mathbb{R}^{p \times p}$ is very small and X is typically well conditioned, both approaches are equally accurate. In terms of performance, the solution via Cholesky decomposition (detailed below) is slightly more efficient.

4	$S_i \coloneqq \overline{X}_i^T \overline{X}_i$	(symmetric matrix product)
5	$\overline{b}_i \coloneqq \overline{X}_i^T \overline{y}$	(matrix times vector)
6	$b_i \coloneqq S_i^{-1} \overline{b}_i$	(linear system via Cholesky)

In this paper, we only consider this approach.

2.1 Multiple SNPs

When the six steps for the solution of one OLS are applied to the specific case of Eq. (1), it is possible to take advantage of the structure of X_i and avoid redundant computation.

Plugging $X_i = \left(X_L \middle| X_{Ri}\right)$ into $\overline{X}_i \coloneqq L^{-1} X_i$ (line 2), we obtain

$$\left(\overline{X}_L \middle| \overline{X}_{Ri}\right) \coloneqq \left(L^{-1} X_L \middle| L^{-1} X_{Ri}\right),$$

that is, $\overline{X}_L \coloneqq L^{-1} X_L$, and $\overline{X}_{Ri} \coloneqq L^{-1} X_{Ri}$. These assignments indicate that the quantity $\overline{X}_L$ can be computed once and reused across all the SNPs.

Similarly, for $S_i \coloneqq \overline{X}_i^T \overline{X}_i$ (line 4), we have[1]

$$\left(\begin{array}{c|c} S_{TL} & * \\ \hline S_{BLi} & S_{BRi} \end{array}\right) \coloneqq \left(\begin{array}{c|c} \overline{X}_L^T \overline{X}_L & * \\ \hline \overline{X}_{Ri}^T \overline{X}_L & \overline{X}_{Ri}^T \overline{X}_{Ri} \end{array}\right),$$

[1]The subscript letters $_L$, $_R$, $_T$, and $_B$ stand for Left, Right, Top, and Bottom, respectively.

```
1   LL^T ≔ M
2   X̄_L ≔ L^{-1} X_L ,    ȳ ≔ L^{-1} y
3   S_TL ≔ X̄_L^T X̄_L ,    b̄_T ≔ X̄_L^T y
4   for i in {1, ..., m}
5       X̄_Ri ≔ L^{-1} X_Ri
6       S_BLi ≔ X̄_Ri^T X̄_L
7       S_BRi ≔ X̄_Ri^T X̄_Ri
8       b̄_Bi ≔ X̄_Ri^T ȳ
9       set S_i ≔ ( S_TL | *  ) ,  b̄_i ≔ ( b̄_T  )
                  ( S_BLi | S_BRi )       ( b̄_Bi )
10      b_i ≔ S_i^{-1} b̄_i
11  end
```

Algorithm 1: Optimized algorithm for the solution of Eq. (1).

from which

$$\begin{aligned} S_{TL} &\coloneqq \overline{X}_L^T \overline{X}_L \in \mathbb{R}^{(p-1)\times(p-1)}, \\ S_{BLi} &\coloneqq \overline{X}_{Ri}^T \overline{X}_L^T \in \mathbb{R}^{1\times(p-1)}, \text{ and} \\ S_{BRi} &\coloneqq \overline{X}_{Ri}^T \overline{X}_{Ri}^T \in \mathbb{R}, \end{aligned}$$

indicating that S_{TL}, the top left portion of S_i, is independent of i and needs to be computed only once.[2] Finally, the same idea applies also to $\overline{b}_i$ (line 5), yielding the assignments $\overline{b}_T \coloneqq \overline{X}_L^T y$ and $\overline{b}_{Bi} \coloneqq \overline{X}_{Ri} y$.

The computation for the whole Eq. (1) is given in Algorithm 1. There, all the operations independent of i are moved outside the loop, thus lowering the overall complexity from $O(n^3 + mn^2 p)$ down to $O(n^3 + mn^2)$.[3] This algorithm constitutes the basis for the large-scale versions presented in the next two sections.

3. OUT-OF-CORE

GWA studies often operate on and generate datasets that exceed the main memory capacity of current computers. For instance, a study with n = 20,000 individuals, m = 10,000,000 SNPs, and $p = 4$, requires 1.49 TB to store the input data (M and X_i's), and generates 305 MB of output.[4] To make large analyses feasible, regardless of the number of SNPs, Fabregat et al. proposed an extended version of Algorithm 1, described below, that streams X_{Ri} and b_i from secondary storage, by means of asynchronous I/O operations [3].

In order to avoid any overhead, the vectors X_{Ri} (and b_i) are grouped into blocks X_{blk} (and b_{blk}) of size m_{blk}, and read (written) asynchronously using double buffering. The idea is to logically split the main memory in two equal regions: One

[2]Since S_i is symmetric, its top-right and bottom-left quadrants are the transpose of each other; we mark the top-right quadrant with a $*$, indicating that it is never accessed nor computed.

[3] Since in most analyses $m \gg n$, the complexity reduces by a factor of p, from $O(mn^2 p)$ down to $O(mn^2)$.

[4]In practice the size of the output is even larger, because in addition to b_i, a $p \times p$ symmetric matrix is also generated.

1 $LL^T \coloneqq M$
2 $\overline{X}_L \coloneqq L^{-1}X_L\,, \quad \overline{y} \coloneqq L^{-1}y$
3 $S_{TL} \coloneqq \overline{X}_L^T\overline{X}_L\,, \quad \overline{b}_T \coloneqq \overline{X}_L^T y$
4 **load_start** first X_{blk}
5 **for each** blk
6 **load_wait** current X_{blk}
7 **if not** last blk: **load_start** next X_{blk}
8 $\overline{X}_{blk} \coloneqq L^{-1}X_{blk}$
9 **for** i **in** $\{1,\ldots,m_{blk}\}$
10 **set** $\overline{X}_{Ri} \coloneqq \overline{X}_{blk}[i]$
11 $S_{BLi} \coloneqq \overline{X}_{Ri}^T\overline{X}_L\,, \quad S_{BRi} \coloneqq \overline{X}_{Ri}^T\overline{X}_{Ri}$
12 $\overline{b}_{Bi} \coloneqq \overline{X}_{Ri}^T\overline{y}$
13 **set** $S_i \coloneqq \left(\begin{array}{c|c} S_{TL} & * \\ \hline S_{BLi} & S_{BRi} \end{array}\right)\,, \quad \overline{b}_i \coloneqq \left(\begin{array}{c} \overline{b}_T \\ \hline \overline{b}_{Bi} \end{array}\right)$
14 $b_i \coloneqq S_i^{-1}\overline{b}_i$
15 **set** $b_{blk}[i] \coloneqq b_i$
16 **end**
17 **if not** first blk: **store_wait** previous b_{blk}
18 **store_start** current b_{blk}
19 **end**
20 **store_wait** last b_{blk}

Algorithm 2: Out-of-core version of Algorithm 1: The X_{Ri} and b_i are streamed from and to disk in blocks. Asynchronous I/O operations are in green.

region is devoted to the block of data that is currently processed, while the other is used to store the output from the previous block and to load the input for the next one. Once the computation on the current block is completed, the roles of the two regions are swapped. The algorithm commences by loading the first block of SNPs X_{blk} from disk into memory; then, while the GLS's corresponding to this block are solved, the next block of SNPs is loaded asynchronously in the second memory region. (Analogously, the previous b_{blk} is stored, while the current one is computed.)

When dealing with large analyses, an important optimization comes from, whenever possible, processing multiple SNPs at once: Algorithm 2 shows how slow vector operations on X_{Ri} can be combined together, originating efficient matrix operations on $X_{blk} \in \mathbb{R}^{n\times m_{blk}}$.

3.1 Shared memory implementation

The implementation of Algorithm 2, called SMP-OOC, makes use of parallelism in two different ways [3]. The operations in lines 1 through 8 are dominated by BLAS3 and take full advantage of a multithreaded implementation of BLAS (LAPACK). By contrast, the operations within the innermost loop (lines 11 through 14), only involve very small or thin matrices, for which BLAS and especially multithreaded BLAS are less efficient. Therefore, thery are scheduled in parallel using OpenMP in combination with single-threaded BLAS and LAPACK.

We compiled SMP-OOC, written in C, with the GNU C compiler (version 4.4.5) and linked to Intel's Math Kernel Library (MKL version 10.3). All tests were executed on a system consisting of two six-core Intel X5675 processors,

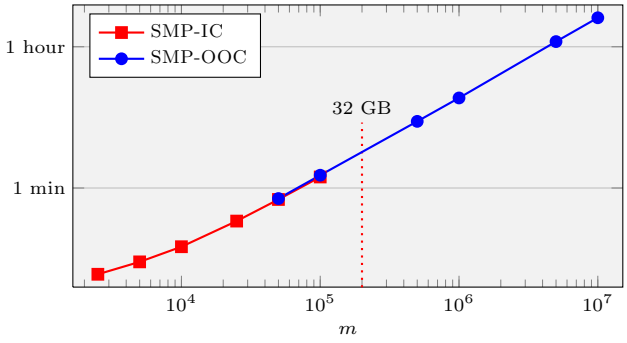

Figure 1: Performance of SMP-IC and SMP-OOC for increasing m. The vertical line is the limit for the in-core version imposed by the RAM size. $n = 10{,}000$, $p = 4$.

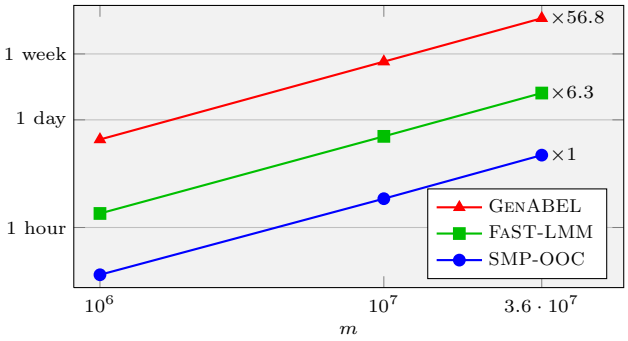

Figure 2: Performance of SMP-OOC compared to GenABEL and FaST-LMM. $n = 10{,}000$, $p = 4$.

running at 3.06 GHz, equipped with 32 GB of RAM, and connected to a 1 TB hard disk.

Preliminary measurements have shown that changing $p \in \{1,\ldots,20\}$ results in performance variation on the order of system fluctuations (below 1%). We therefore consider $p = 4$, a value encountered in several GWA studies, throughout all our experiments.

In the first experiment, we compare the efficiency of SMP-OOC with SMP-IC, an equivalent in-core version. We fixed $n = 10{,}000$, $p = 4$, and we let m vary between 10^3 and 10^7. For the out-of-core version, the SNPs were grouped in blocks of size $m_{blk} = 5{,}000$. Figure 1 shows that SMP-OOC scales linearly in the number of SNPs, well beyond the maximum problem size imposed by the 32 GB of RAM. Furthermore, the fact that the lines for the in-core and out-of-core algorithms overlap perfectly confirms that the I/O operation to and from disk are entirely hidden by computation.

In the second experiment, Figure 2, we show the performance of SMP-OOC with respect to that of other two solvers: FaST-LMM, a program designed for GWAS on large datasets [8] and GenABEL, a widely spread library for genome studies [1]. Again, we fixed $n = 10{,}000$ and $p = 4$, while m varies between 10^6 and $3.6 \cdot 10^7$. The fairly constant observed speedups of SMP-OOC over FaST-LMM and GenABEL are, at $m = 3.6 \cdot 10^7$, 6.3 and 56.8, respec-

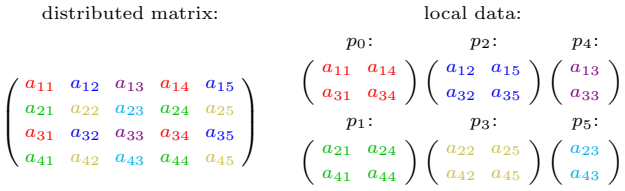

Figure 3: Default 2D matrix distribution on a 2×3 process grid.

tively.

4. DISTRIBUTED MEMORY

While SMP-OOC scales up to an arbitrarily large amount of SNPs m, the main memory is still a limiting factor for the population size n: In fact, the algorithm necessitates the matrix $M \in \mathbb{R}^{n \times n}$ (or equivalently, its Cholesky factor L) to reside fully in memory. Due to the triangular solve (Algorithm 2, line 2), keeping the matrix in the secondary storage is not a viable option. Our approach here consists in distributing M, L, and all matrices on which L operates, across multiple compute nodes, lifting any constraint on their size.

4.1 Elemental

As a framework for distributed-memory dense linear algebra operations, we use Elemental [10]. This is a C++ library, based on the Message Passing Interface (MPI), that operates on a virtual two-dimensional grid of processes; the name is inspired by the fact that, in general, matrices are cyclically distributed across a 2D grid of processes in an element-wise fashion. This principal distribution[5] is shown in Figure 3;

Algebraic operations on distributed matrices involve two stages: data redistribution (communication), and invocation of single-node BLAS or LAPACK routines (computation). Optimal performance is attained by minimizing communication within the redistributions. In most cases, as shown in [10], this is achieved by choosing the process grid to be as close to a perfect square as possible.

While a square process grid is optimal for performance, since all processes only hold non-contiguous portions of the matrix, it complicates loading contiguously stored data from files into a distributed matrix. In the context of GWAS, the algorithm has to load two objects of different nature: the matrix M, and the collections of vectors X_{blk}; the special nature of the latter determines that the vectors can be loaded and processed in any order.

For loading M, we first read contiguous panels into the local memory of each process via standard file operations, and then construct the global (distributed) version of M by accumulating the panels. This is done via Elemental's axpy-interface, a feature that makes it possible to add node-local matrices to a global one.

For loading X_{blk} instead, a collection of contiguously stored vectors is read into memory through more efficient means

[5]In Elemental's notation: $[MC, MR]$.

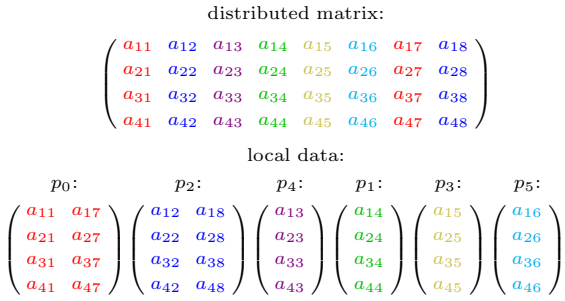

Figure 4: 1D matrix distribution on a 1×6 process grid.

than the axpy-interface by exploiting that, as long as consistently handled, the order of the vectors is irrelevant. The trick is to use a matrix that is distributed on a virtual 1D reordering of the grid into a row of processes. As shown in Figure 4, the process-local data of such a matrix is a set of full columns, which can be loaded from a contiguous datafile. While these local columns are not adjacent in the distributed matrix, Elemental guarantees that all algebraic operations performed on them maintain their order. For performance reasons, prior to any computation, the matrix on the 1D ordering of this grid needs to be redistributed to conform to the initial 2D process grid (Figure 3). This redistribution, provided by Elemental, can internally be performed most efficiently through a single `MPI_Alltoall` if the 1D grid is the concatenation of the rows of the 2D grid.[6]

4.2 The parallel algorithm

In Algorithm 3, we present the distributed-memory version of Algorithm 2 for np processes; the matrices that are distributed among the processes and the corresponding operations are highlighted in blue; the quantities that differ from one process to another are instead in red.

The algorithm begins (line 1) by loading the first $\frac{m_{blk}}{np}$ vectors X_{Ri} into a local block X_{blk} on each process asynchronously. It commences with the initially distributed M, X_L, and y, and computes L, $\overline{X}_L$, and $\overline{y}$ (lines 2 – 3). Then, X_L and y, local copies of X_L and y, respectively, are created on each process (line 4). Since small local computations are significantly more efficient than the distributed counterparts, S_{TL} and b_T are computed redundantly by all processes (line 5).

In line 9, the asynchronously loaded blocks X_{blk} are — without any communication or memory transfers— seen as the columns of X_{blk} that are cyclically distributed across a 1D process grid as described in Section 4.1. Since in Elemental matrix operations require all operands to be in the default distribution across the 2D grid, X_{blk} and $\overline{X}_{blk}$ are redistributed before and after the computation in line 10, respectively. Once $\overline{X}_{blk}$ is computed and redistributed, in line 11, each process views its local columns of this matrix as $\overline{X}_{blk}$; since the distributions of X_{blk} and $\overline{X}_{blk}$ are identical, these —without communication or transfers— correspond to the columns of X_{blk}.

[6]In Elemental: $[*, VR]$.

1 **load_start** first X_{blk}
2 $LL^T := M$
3 $\overline{X}_L := L^{-1}X_L$, $\overline{y} := L^{-1}y$
4 **copy** $\overline{X}_L := \overline{X}_L$, $\overline{y} := \overline{y}$
5 $S_{TL} := \overline{X}_L^T\overline{X}_L$, $\overline{b}_T := \overline{X}_L^T y$
6 **for each** blk
7 **load_wait** current X_{blk}
8 **if not** last blk: **load_start** next X_{blk}
9 **set** $X_{blk} :=$ combine(X_{blk})
10 $\overline{X}_{blk} := L^{-1}X_{blk}$
11 **set** $\overline{X}_{blk} :=$ localpart$(\overline{X}_{blk})$
12 $S_{blk} := \overline{X}_{blk}{}^T\overline{X}_L$
13 **for** i **in** $\{1, \ldots, \frac{m_{blk}}{np}\}$
14 **set** $\overline{X}_{Ri} := \overline{X}_{blk}[i]$, $S_{BLi} := S_{blk}[i]$
15 $S_{BRi} := \overline{X}_{Ri}{}^T\overline{X}_{Ri}$
16 $\overline{b}_{Bi} := \overline{X}_{Ri}{}^T\overline{y}$
17 **set** $S_i := \left(\begin{array}{c|c} S_{TL} & * \\ \hline S_{BLi} & S_{BRi} \end{array}\right)$, $\overline{b}_i := \left(\begin{array}{c} \overline{b}_T \\ \hline \overline{b}_{Bi} \end{array}\right)$
18 $b_i := S_i^{-1}\overline{b}_i$
19 **set** $b_{blk}[i] := b_i$
20 **end**
21 **if not** first blk: **store_wait** previous b_{blk}
22 **store_start** current b_{blk}
23 **end**
24 **store_wait** last b_{blk}

Algorithm 3: Distributed memory version of Algorithm 1. Asynchronous I/O operations are depicted green, distributed matrices and operations in blue, and quantities that differ across processes in red.

In addition to blocking X_{Ri} and b_{Bi}, the computation of all row vectors S_{BLi} belonging to the current block is combined into a single matrix product (line 12) resulting in the S_{BLi} being stacked in a block S_{blk}. In line 14, S_{BLi} is selected from S_{blk}, along with X_{Ri} from X_{blk} for the innermost loop. This loop then computes the local b_{blk} independently on each process. Finally, b_{blk} (whose columns b_i corresponds to the initially loaded vectors X_{Ri} within X_{blk}) is stored asynchronously, while the next iteration commences.

4.3 Performance Results

We compile Elem-OOC, the C++-implementation of Algorithm 3, with the GNU C compiler (version 4.7.2), use Elemental (version 0.78-dev) with OpenMPI (version 1.6.4) and link to Intel's Math Kernel Library (MKL version 11.0). In our tests, we use a compute cluster with 40 nodes, each equipped with 16 GB of RAM and two quad-core Intel Harpertown E5450 processors running at 3.00 Ghz. The nodes are connected via InfiniBand and access a high speed Lustre file system.

Throughout all our experiments, we use the empirically optimal local block-size $\frac{m_{blk}}{np} = 256$ by choosing $m_{blk} = 256np$.

4.3.1 *Processing huge numbers of SNPs out-of-core*

Since Elem-OOC incorporates the double-buffering method introduced in Section 3, it can process datasets with arbitrarily large m without introducing any overhead due to I/O operations. To confirm this claim, we perform a series of experiments, using $np = 8$, 16, 32, and 64 cores (1, 2, 4, and 8 nodes) to solve a system of size $n = 40{,}000$ and $p = 4$ with increasing dataset size m. The performance of these experiments is presented in Figure 5, where the vertical lines mark the points at which the 16 GB of RAM per node are insufficient to store all m vectors X_{Ri}. The plot shows a very smooth behavior with m (dominated by the triangular solve in Algorithm 3, line 10) well beyond this in-core memory limit.

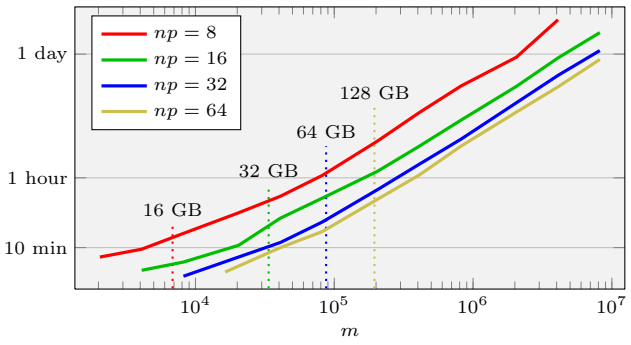

Figure 5: Performance of Elem-OOC as a function of m. Here, $n = 40{,}000$, $p = 4$, and m ranges from 2,048 to $8.2 \cdot 10^6$. The vertical lines are limits for a theoretical in-core version of the parallel algorithm imposed by the accumulated RAM sizes.

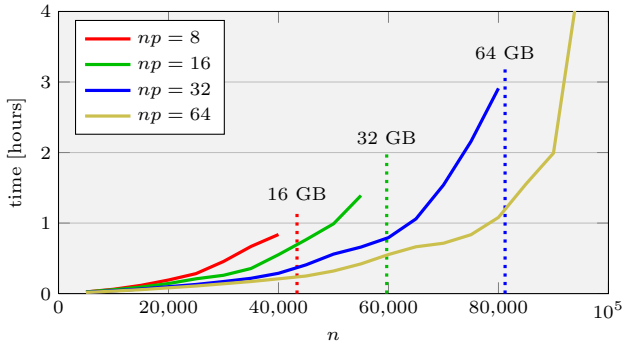

Figure 6: Performance of Elem-OOC as a function of n. $p = 4$, $m = 65{,}536$, and n ranges from 5,000 to 100,000. The vertical lines indicate the limits imposed by the accumulated RAM sizes.

4.3.2 *Increasing the population size n*

We now turn to the main goal of our effort: performing computations on systems whose matrix $M \in \mathbb{R}^{n \times n}$ exceeds the capacity of the main memory. For this purpose, we use $m = 65{,}536$, $p = 4$ and execute Elem-OOC on $np = 8$, 16, 32, and 64 cores (1, 2, 4, and 8 nodes) with increasing matrix size n. Figure 6 reports the performance of these executions, which is dominated by the cubic complexity of the Cholesky factorization of M (Algorithm 3, line 2). The vertical lines indicate where the nodes' memory would be exceeded by the size of the distributed M and the buffers for X_{blk}. The plot shows that our implementation succeeds

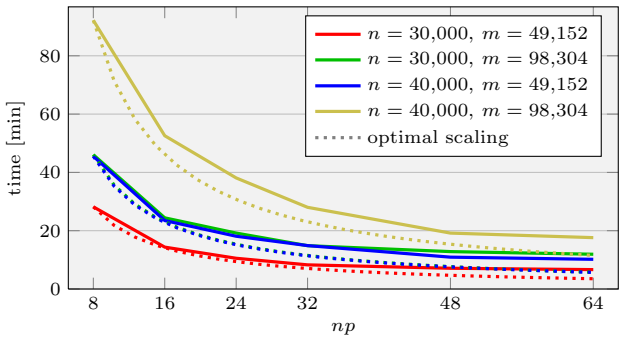

Figure 7: Performance of ELEM-OOC as a function of np. $p = 4$, and np ranges from 8 to 64.

in overcoming these memory limitations through increasing the number of nodes.

4.3.3 *Strong scalability*

In practice, the problem sizes are bound to the specific GWAS and the interest lies in solving Eq. (1) as fast as possible. In the following experiment, we investigate how the time to solution is reduced by ELEM-OOC through increasing the number of compute units, while keeping the problem size constant. In Figure 7, we present the performance attained for four different problem sizes with 8 up to 64 cores (1 through 8 nodes). It shows perfect scalability for increasing the number of processes from 8 to 16, reducing the runtime by a factor of 2. With even more processes, the parallel efficiency decreases, since the local portions of L become too small, but execution time is reduced further.

5. CONCLUSION

We presented two parallel algorithms for solving the generalized least squares problems that arise in genome-wide association studies (GWAS). They address the issue of growing dataset sizes due to the number of studied polymorphisms m and/or the population size n.

The first algorithm uses a double buffering technique in order to process datasets with arbitrarily large numbers of genetic polymorphisms. Compared to other wide-spread GWAS-codes, this algorithm's shared memory implementation, SMP-OOC, was shown to be at least one order of magnitude faster.

The second algorithm enables the processing of datasets involving large populations by storing the covariance matrix in the combined main memory of distributed memory architectures. ELEM-OOC, the implementation of this algorithm, was shown to scale very well in both the population size and the number of processes used.

Together, these two algorithms form a viable basis for the challenges posed by the scale of current and future genome-wise association studies.

5.1 Future Work

The work presented in this paper can be extended in several ways.

- Hybrid parallelism, i.e., using multithreaded BLAS and LAPACK, as well as OPENMP, offers further potential to boost the performance and efficiency of our distributed memory implementation ELEM-OOC.

- When a GWAS is interested in more than one trait y, a further dimension j is added to the set of generalized least squares problems in Eq. (1):

$$b_{ij} = \left(X_i^T M_j^{-1} X_i\right)^{-1} X_i^T M_j^{-1} y_j$$

  with $i = 1, \ldots, m$ and $j = 1, \ldots, t$. A highly efficient shared memory implementation for this problem is already presented in [4]; only a distributed memory implementation on the lines of ELEM-OOC would be capable of solving this problem for large population sizes.

- Since the covariance matrix M represents the relatedness of a diverse population, its few significant entries can be grouped close to the diagonal. This allows to significantly reduce computation time by operating on banded matrices.

6. REFERENCES

[1] Y. S. Aulchenko, S. Ripke, A. Isaacs, and C. M. van Duijn. GenABEL: an R library for genome-wide association analysis. *Bioinformatics*, 23(10):1294–1296, May 2007.

[2] E. Boerwinkle, R. Chakraborty, and C. F. Sing. The use of measured genotype information in the analysis of quantitative phenotypes in man. I. Models and analytical methods. *Ann. Hum. Genet.*, 50(Pt 2):181–194, May 1986.

[3] D. Fabregat-Traver, Y. S. Aulchenko, and P. Bientinesi. Solving sequences of generalized least-squares problems on multi-threaded architectures. *CoRR*, abs/1210.7325, 2012.

[4] D. Fabregat-Traver and P. Bientinesi. Computing petaflops over terabytes of data: The case of genome-wide association studies. *CoRR*, abs/1210.7683, 2012. Accepted at ACM TOMS.

[5] L. A. Hindorff, J. MacArthur, J. Morales, H. A. Junkins, P. N. Hall, A. K. Klemm, and T. A. Manolio. A catalog of published genome-wide association studies. http://www.genome.gov/gwastudies/. Accessed: Mar 2013.

[6] L. A. Hindorff, P. Sethupathy, H. A. Junkins, E. M. Ramos, J. P. Mehta, F. S. Collins, and T. A. Manolio. Potential etiologic and functional implications of genome-wide association loci for human diseases and traits. *Proc. Natl. Acad. Sci. U.S.A.*, 106(23):9362–9367, Jun 2009.

[7] R. J. Klein, C. Zeiss, E. Y. Chew, J.-Y. Tsai, R. S. Sackler, C. Haynes, A. K. Henning, J. P. SanGiovanni, S. M. Mane, S. T. Mayne, M. B. Bracken, F. L. Ferris, J. Ott, C. Barnstable, and J. Hoh. Complement factor h polymorphism in age-related macular degeneration. *Science*, 308(5720):385–389, 2005.

[8] C. Lippert, J. Listgarten, Y. Liu, C. M. Kadie, R. I. Davidson, and D. Heckerman. Fast linear mixed models for genome-wide association studies. *Nature Methods*, 8(10):833–835, 2011.

[9] T. A. Manolio. Published gwas reports, 2005 – 6/2012. http://www.genome.gov/multimedia/illustrations/Published_GWA_Reports_6-2012.pdf.

[10] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2):13:1–13:24, Feb. 2013.

[11] J. Yu, G. Pressoir, W. H. Briggs, I. Vroh Bi, M. Yamasaki, J. F. Doebley, M. D. McMullen, B. S. Gaut, D. M. Nielsen, J. B. Holland, S. Kresovich, and E. S. Buckler. A unified mixed-model method for association mapping that accounts for multiple levels of relatedness. *Nat. Genet.*, 38(2):203–208, Feb 2006.